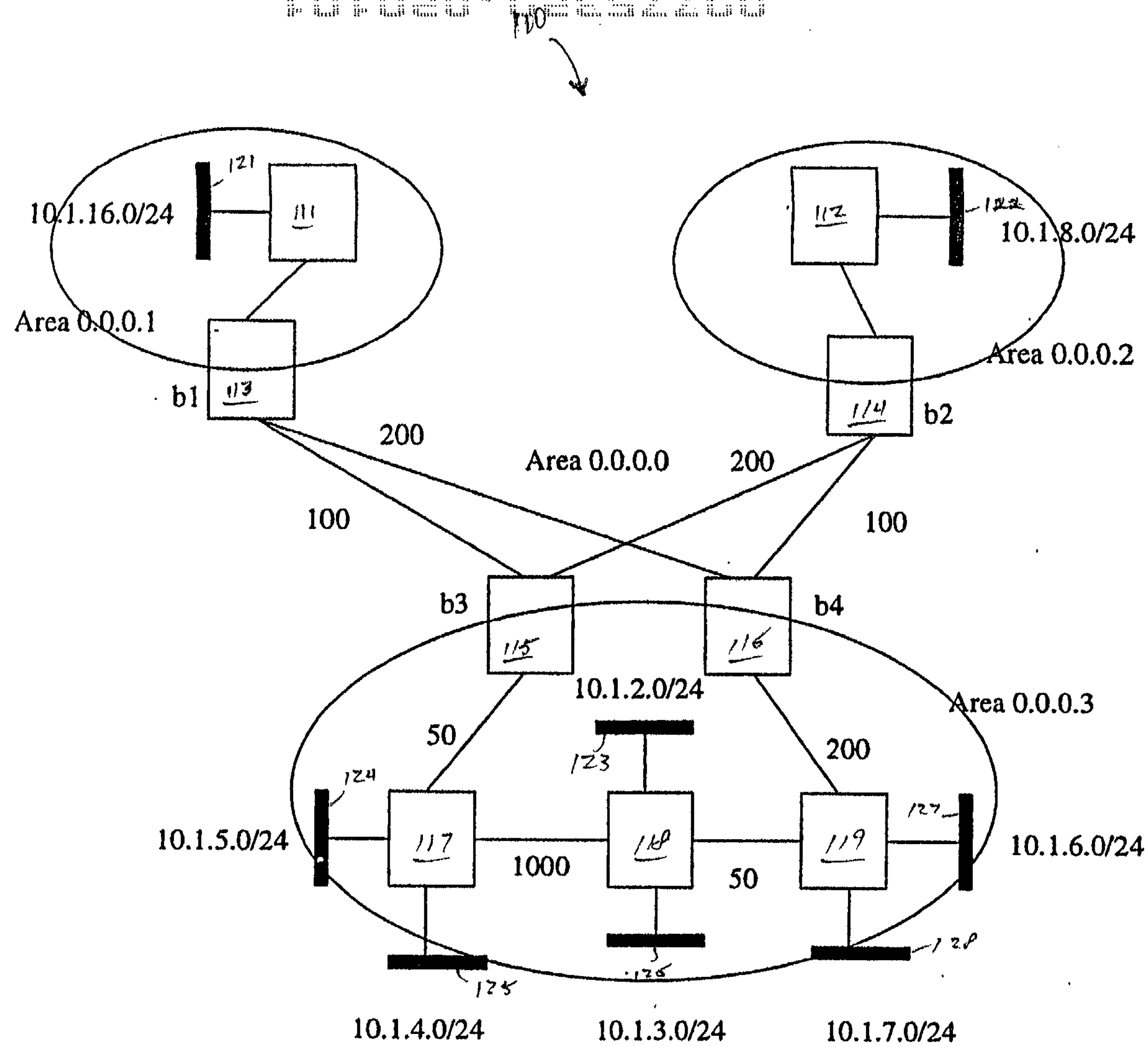

100
10.1.16.0/24
121
111
Area 0.0.0.1
b1
113
112
122
10.1.8.0/24
Area 0.0.0.2
114
b2
200
Area 0.0.0.0
200
100
100
b3
115
b4
116
10.1.2.0/24
Area 0.0.0.3
50
123
200
124
10.1.5.0/24
117
1000
118
50
119
127
10.1.6.0/24
125
126
128
10.1.4.0/24
10.1.3.0/24
10.1.7.0/24

FIGURE 1

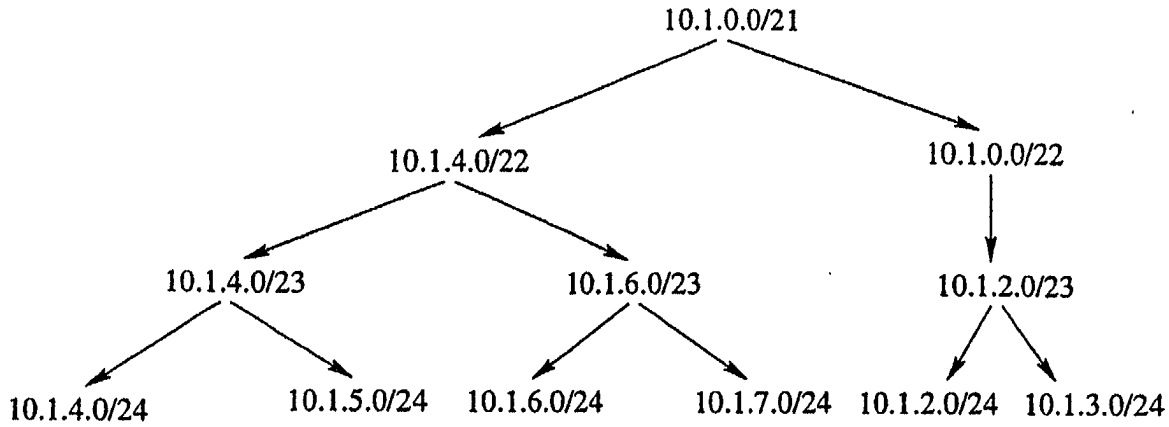

10.1.0.0/21
10.1.4.0/22
10.1.0.0/22
10.1.4.0/23
10.1.6.0/23
10.1.2.0/23
10.1.4.0/24
10.1.5.0/24
10.1.6.0/24
10.1.7.0/24
10.1.2.0/24
10.1.3.0/24

FIGURE 2

```
procedure ComputeMinError(Aggregate x, Aggregate y, integer l)
1.  if subTree[x, y, l].computed = true
2.     return [subTree[x, y, l].error, subTree[x, y, l].aggregates]
3.  minError := minError1 := minError2 := ∞
4.  if x is a leaf {
5.     minError1 := ∑_{s∈S} D(s,t) * (lsp(s, x, {y}, W_A) − lsp(s, x))
6.     if l > 0
7.        minError2 := ∑_{s∈S} D(s,t) * (lsp(s, x, {x}, W_A) − lsp(s, x))
8.     if minError1 ≤ minError2
9.        [subTree[x, y, l].error, subTree[x, y, l].aggregates] := [minError1, ∅]
10.    else
11.       [subTree[x, y, l].error, subTree[x, y, l].aggregates] := [minError2, {x}]
12. }
13. if x has a single child u {
14.    [minError1, aggregates1] := ComputeMinError(u, y, l)
15.    if l > 0
16.       [minError2, aggregates2] := ComputeMinError(u, x, l − 1)
17.    if minError1 ≤ minError2
18.       [subTree[x, y, l].error, subTree[x, y, l].aggregates] := [minError1, aggregates1]
19.    else
20.       [subTree[x, y, l].error, subTree[x, y, l].aggregates] := [minError2, aggregates2 ∪ {x}]
21. }
22. if x has children u and v {
23.    for i := 0 to l {
24.       [minError1, aggregates1] := ComputeMinError(u, y, i)
25.       [minError2, aggregates2] := ComputeMinError(v, y, k − i)
26.       if minError1 + minError2 < minError
27.          minError := minError1 + minError2
28.          aggregates := aggregates1 ∪ aggregates2
29.    }
30.    for i := 0 to l − 1 {
31.       [minError1, aggregates1] := ComputeMinError(u, x, i)
32.       [minError2, aggregates2] := ComputeMinError(v, x, k − i − 1)
33.       if minError1 + minError2 < minError
34.          minError := minError1 + minError2
35.          aggregates := aggregates1 ∪ aggregates2 ∪{x}
36.    }
37.    [subTree[x, y, l].error, subTree[x, y, l].aggregates] := [minError, aggregates]
38. }
39. subTree[x, y, l].computed := true
40. return [subTree[x, y, l].error, subTree[x, y, l].aggregates]
```

FIGURE 3

```
procedure CombineMinError()
1.  for i = 1 to m
2.      for j = 0 to k {
3.          T_i[j].[error, aggregates] := ComputeMinError(r(T_i), ε, j)
4.          X_i[j].[error, aggregates] := [∞, ∅]
5.      }
6.  for j = 0 to k
7.      X_1[j].[error, aggregates] := T_1[j].[error, aggregates]
8.  for i = 1 to m
9.      for j = 0 to k
10.         for l = 0 to j
11.             if (X_{i-1}[l].error + T_i[j - l].error < X_i[j].error) {
12.                 X_i[j].error = X_{i-1}[l].error + T_i[j - l].error
13.                 X_i[j].aggregates = X_{i-1}[l].aggregates ∪ T_i[j - l].aggregates
14.             }
```

FIGURE 4

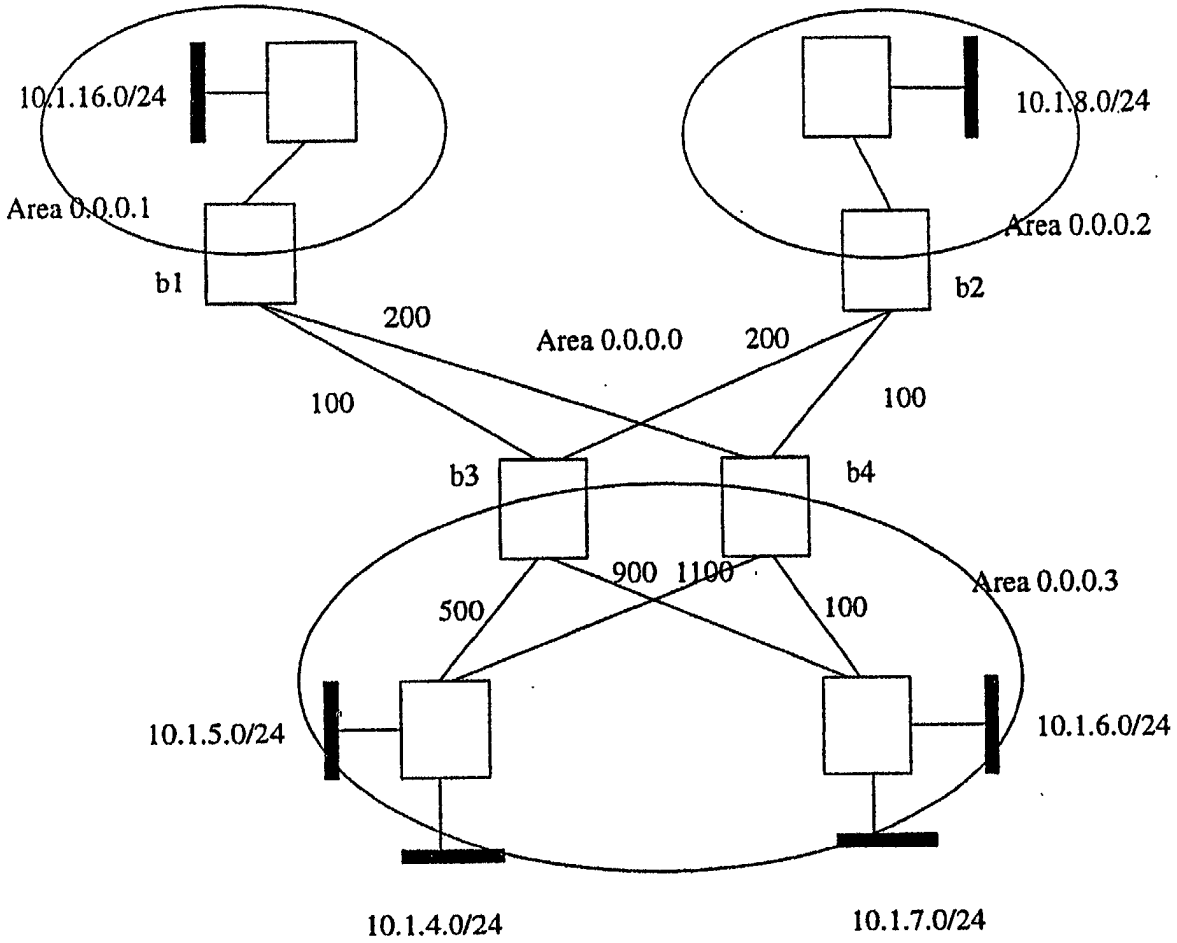

10.1.16.0/24
10.1.8.0/24
Area 0.0.0.1
Area 0.0.0.2
b1
b2
200
Area 0.0.0.0
200
100
100
b3
b4
900
1100
Area 0.0.0.3
500
100
10.1.5.0/24
10.1.6.0/24
10.1.4.0/24
10.1.7.0/24

FIGURE 5

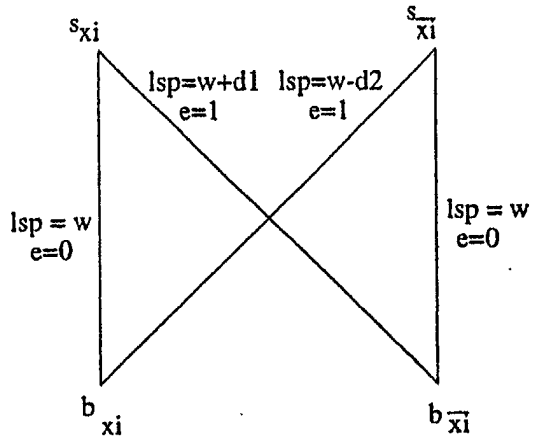
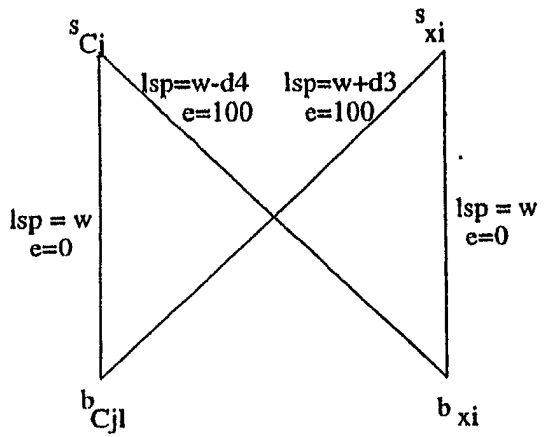
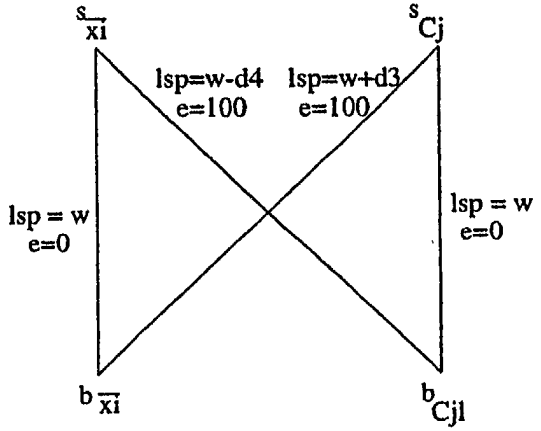


FIGURE 6



(a) $C_{jl} = x_i$

FIGURE 7A



(b) $C_{jl} = \bar{x}_i$

FIGURE 7B

```
procedure ComputeWeightsCumulative()
1. for each b ∈ B_i set W_min(b) := 0
2. for i := 1 to r {
3.    W := W_min
4.    Choose a random subset R ⊆ B_i of ABRs
5.    for each b ∈ R set W(b) to a random weight in [0, L]
6.    if Σ_{s∈S} e(s, B(s, W)) < Σ_{s∈S} e(s, B(s, W_min))
7.       W_min := W
8. }
9. return W_min
```

FIGURE 8

procedure ComputeWeightsMax(Q)

1. for each b ∈ B_i set Wold(b) := 0
2. while (Pb_2B

i Wold(b) ≤ (

j B_i j*(j B_i j-1)

2) *lspmax) f3. Let

Q0 be a new set of inequalities that result when the value Wold(b) is substituted for each variable W (b)only on the LHS of each inequality in Q 4. Set Wnew(b) to the smallest possible value such that each inequality in Q0 is satisfied when Wnew(b) is substituted for variable W (b) in Q0 5. if Wnew = Wold 6. return Wnew 7. else 8. Wold := Wnew 9. g 10. return "there does not exist a weight assignment W "

FIGURE 9

```
procedure ComputeWeightsTwoABR()
1. Set V_opt := v(s_1), E := E_opt := Σ_{s∈S} e(s, b_1)
2. for j := 1 to n {
3.     E := E + e(s_j, b_2) − e(s_j, b_1)
4.     if E < E_opt
5.         V_opt := v(s_{j+1}), E_opt := E
6. }
7. return V_opt
```

FIGURE 10